

## Laboratorio Nro. 3

### Listas Enlazadas (*Linked List*) y Listas Hechas con Arreglos (*Array List*)

#### Objetivos

1. Entender cómo se utiliza en un lenguaje de programación para resolver problemas las implementaciones de listas implementadas con arreglos (*ArrayList*) y listas doblemente enlazadas (*LinkedList*)
2. Implementar listas enlazadas
3. Usar herramientas para probar software
4. Utilizar listas, colas y pilas para solucionar problemas

#### Consideraciones Iniciales

##### Leer la Guía



Antes de comenzar a resolver el presente laboratorio, leer la **“Guía Metodológica para la realización y entrega de laboratorios de Estructura de Datos y Algoritmos”** que les orientará sobre los requisitos de entrega para este y todos los laboratorios, las rúbricas de calificación, el desarrollo de procedimientos, entre otros aspectos importantes.

##### Registrar Reclamos



En caso de tener **algún comentario** sobre la nota recibida en este u otro laboratorio, pueden **enviarlo** a través de <http://bit.ly/2g4TTKf>, el cual será atendido en la menor brevedad posible.

**Traducción de Ejercicios** En el GitHub del docente, encontrarán la traducción al español de los enunciados de los Ejercicios en Línea.



**Visualización de  
Calificaciones**



A través de **Eafit Interactiva** encontrarán un **enlace** que les permitirá **ver un registro de las calificaciones** que **emite el docente** para cada taller de laboratorio y según las rubricas expuestas. **Véase sección 3, numeral 3.7.**

**GitHub**

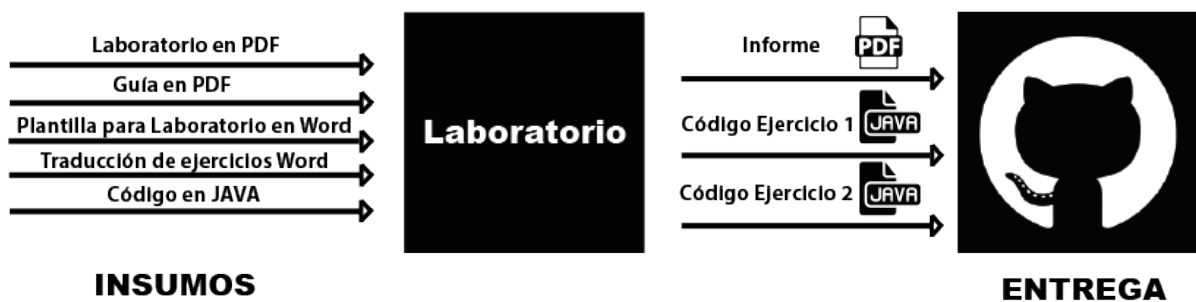


Crear un repositorio en su cuenta de GitHub con el nombre `st0245-suCodigoAqui`. **2.** Crear una carpeta dentro de ese repositorio con el nombre `laboratorios`. **3.** Dentro de la carpeta `laboratorios`, crear una carpeta con nombre `lab03`. **4.** Dentro de la carpeta `lab03`, crear tres carpetas: `informe`, `codigo` y `ejercicioEnLinea`. **5.** Subir el informe pdf a la carpeta `infome`, el código del ejercicio 1 a la carpeta `codigo` y el código del ejercicio en línea a la carpeta `ejercicioEnLinea`. Así:

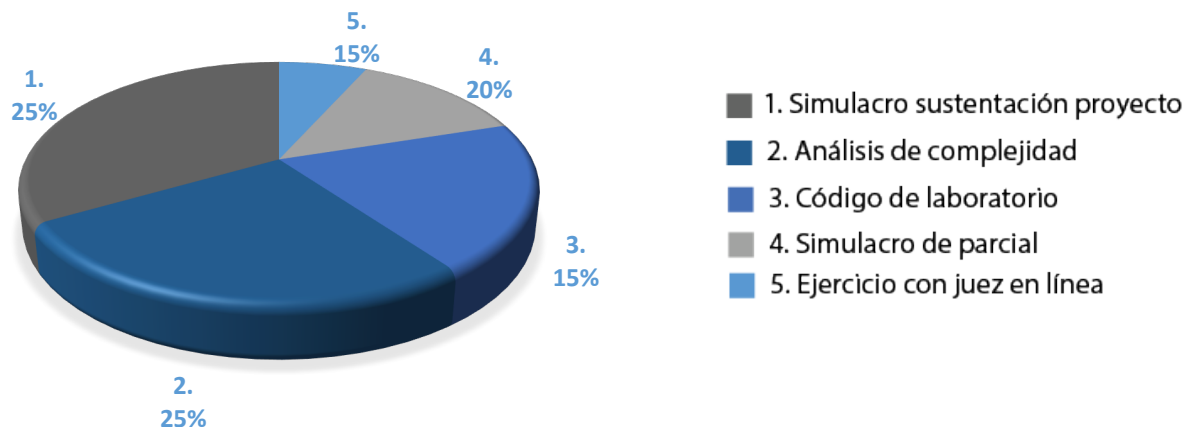
```
st0245-suCodigoAqui
  laboratorios
    lab01
      informe
      codigo
      ejercicioEnLinea
    lab02
    ...
```

## Intercambio de archivos

Los archivos que **ustedes deben entregar** al docente son: **un archivo PDF** con el informe de laboratorio usando la plantilla definida, y **dos códigos**, uno con la solución al numeral 1 y otro al numeral 2 del presente. Todo lo anterior se entrega en **GitHub**.



## Porcentajes y criterios de evaluación para el laboratorio



## Ejercicios a resolver

### 1. Códigos para entregar en GitHub:



En la vida real, la documentación del software hace parte de muchos estándares de calidad como CMMI e ISO/IEC 9126



Véase Guía **en Sección 3, numeral 3.4**



Código de laboratorio en **GitHub**. Véase Guía en **Sección 4, numeral 4.24**



*Entregar documentación en **JavaDoc** o equivalente. El uso de JavaDoc es opcional.*



**No se reciben** archivos en **.RAR** ni en **.ZIP**

Para los ejercicios 1.1 al 1.4, tengan en cuenta que:

- ☒ Deben utilizar la clase *Laboratorio3*. Dentro de esta clase, definir un método para cada ejercicio y en la clase *main*, mostrar con ejemplos cómo funciona cada ejercicio.
- ☒ Si deciden usar otro lenguaje de programación, asegúrense de usar una implementación de listas enlazadas y una de listas con arreglos.

- ☑ En la mayoría de los lenguajes (*Php, Python, Ruby*), las listas, por defecto, están hechas con arreglos.
- ☑ Para cada función implementar un método que trabaje con *ArrayList* y uno que trabaje con *LinkedList* o, si es posible, un método que sirva para los dos tipos de lista (sí es posible). Utilicen las listas disponibles en el API de Java.
- ☑ Utilicen la interfaz *List* de Java <http://bit.ly/2hxCaNu>

**1.1** Desarrollen un método que reciba una lista y devuelva la multiplicación de todos los números introducidos en una lista. Como un ejemplo, para el arreglo [1,2,3,4], el método debe retornar  $1*2*3*4 = 24$ .



**NOTA 1:** No es imprimir en la pantalla, es retonar



En la vida real insertar elementos en una colección sin permitir repetidos se utiliza para implementar la estructura de datos **Conjunto**. La estructura de datos conjunto se encuentra en lenguajes como Python, C++, Java y SQL

**1.2** Teniendo en cuenta lo anterior, implementen un método que reciba una lista e inserte un dato de modo similar a la función insertar al final de la lista. La diferencia es que ahora no se debe permitir insertar datos repetidos. Si un dato ya está almacenado, entonces la lista no varía, pero tampoco es un error. Llámelo `SmartInsert(Lista l, int data)`.



En la vida real, balancear los pesos sobre una viga se conoce como equilibrio estático, y es de interés en Ingeniería Mecánica e Ingeniería Civil

**1.3** Teniendo en cuenta lo anterior, escriban un método que reciba una lista y calcule cuál es la posición óptima de la lista para colocar un pivote



**NOTA 1:** Este algoritmo se puede hacer complejidad  $O(n)$  donde  $n$  es el número de elementos en la lista.



En la vida real, las listas enlazadas se usan para representar objetos en videojuegos (Ver más en <http://bit.ly/2mcGa5w>) y para modelar pistas en juegos de rol (Ver más en <http://bit.ly/2IPyXGC>)

**1.4** Se tiene un almacén donde se encuentran las neveras fabricadas por una planta

- ☒ Las primeras neveras que fueron fabricadas están de últimas dentro del almacén; las últimas neveras fabricadas recientemente, aparecen más cerca de la puerta del almacén.
- ☒ Para sacar una nevera que está atrás, primero habría que quitar la que está adelante.
- ☒ Los datos de cada nevera son el código (representado por un número) y la descripción (representada por un texto).

- ☑ El almacén dispone de una sola puerta por donde las neveras entran las neveras para ser almacenadas y por donde salen las neveras que se van a distribuir a las tiendas.
- ☑ Se tiene una lista de solicitudes de neveras, realizadas por las tiendas, donde aparece el nombre de la tienda y la cantidad solicitada de neveras. Se deben atender primero las solicitudes más antiguas y luego las más recientes.

Teniendo en cuenta lo anterior, elaboren un programa que reciba las neveras, en la forma en que están ordenadas en el almacén, y las solicitudes, según el orden en que llegaron, y que imprima qué neveras del almacén quedan asignadas a cada tienda. Utilice un método así:

```
public static void ejercicio4( ??? neveras, ??? solicitudes)
```



**NOTA 1:** Este algoritmo se puede hacer complejidad  $O(n.m)$  donde  $n$  es el número de solicitudes y  $m$  el máximo número de neveras en una solicitud.

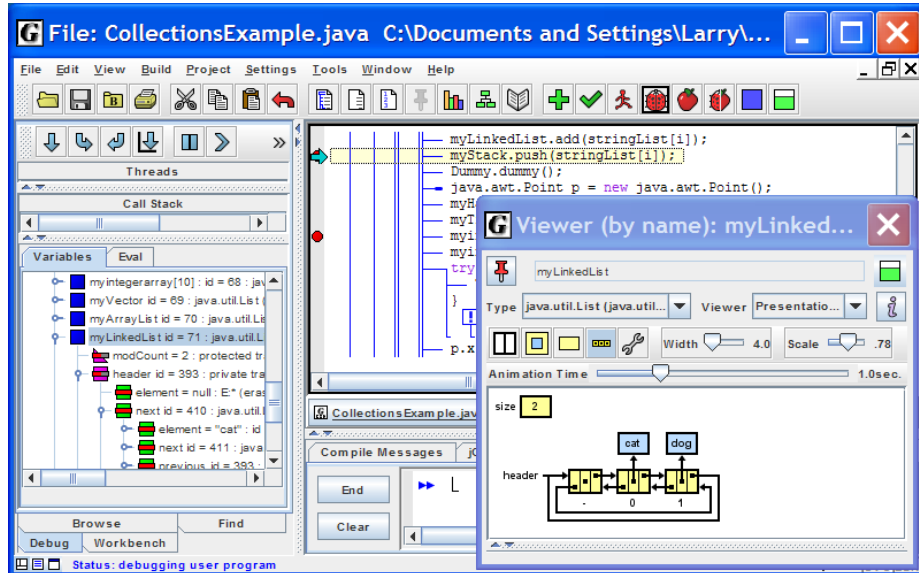
**1.5 Implementen los métodos insertar un elemento en una posición determinada (conocido como *insert* o *add*), borrar un dato en una posición determinada (conocido como *remove*) y verificar si un dato está en la lista (conocido como *contains*) en la clase *LinkedListMauricio*, teniendo en cuenta que sea una lista DOBLEMENTE enlazada.**



**NOTA 1:** Utilice el IDE Jgrasp (<http://www.jgrasp.org/>) porque tiene un depurador gráfico excelente para estructuras de datos.



**NOTA 2:** Véase a continuación gráfica de Jgrasp para efectos de ejemplificación



**1.6** En el código entregado por el profesor está el método *get* y sus pruebas. Teniendo en cuenta esto, realicen 3 tests unitarios para cada método. La idea es probar que su implementación de los métodos *insert* y *remove* funcionan correctamente por los menos en los siguientes casos:

- ☒ Cuando vamos a insertar/borrar y la lista está vacía,
- ☒ Cuando vamos *insertar/borrar* el primer el elemento,
- ☒ Cuando vamos a *insertar/borrar* el último elemento.

**1.7** Teniendo en cuenta lo anterior, resuelvan lo siguiente:

En un banco hay 4 filas de clientes y solamente 2 cajeros. **Se necesita simular cómo son atendidos los clientes por los cajeros.** Si lo desean, usen su implementación de listas enlazadas; de lo contrario, use la del API de Java

- a) Escriban un método que reciba las 4 filas de personas. No se sabe la longitud máxima que puede tener una fila de clientes. Representen las filas de clientes como mejor corresponda.









- b) El método debe hacer una simulación y mostrar en qué cajero (1 o 2) se atiende a cada cliente de cada fila. Coloquen el método dentro de una clase *Banco*. Impriman los datos de la simulación en pantalla.
- c) El cajero uno se identifica con el número 1, el cajero dos con el 2. Los clientes se identifican con su nombre.
- d) El orden en que se atienden los clientes en cada ronda es el siguiente: primero el de la fila 1, luego el de la fila 2, luego el de la fila 3, y finalmente el de la fila 4.

Los cajeros funcionan de la siguiente forma en cada ronda: primero el cajero 1 atiende un cliente, luego el cajero 2 atiende un cliente. Se hacen rondas hasta que no queden más clientes. Si no hay clientes en una fila, se pasa a la siguiente.



**NOTA 1:** Todos los ejercicios del numeral 1 deben ser documentados en formato HTML. Véase *Guía en Sección 4, numeral 4.1 “Cómo escribir la documentación HTML de un código usando JavaDoc”*

## 2) Ejercicios en línea sin documentación HTML en GitHub

	Véase Guía en <b>Sección 3, numeral 3.3</b>		<b>No entregar documentación HTML</b>
	Entregar un archivo en <b>.JAVA</b>		<b>No se reciben archivos en .PDF</b>
	<b>Resolver</b> los problemas de <b>CodingBat</b> usando <b>Recursión</b>		Código del ejercicio en línea en <b>GitHub</b> . Véase Guía en <b>Sección 4, numeral 4.24</b>



**NOTA 1:** Recuerden que, **si toman la respuesta de alguna fuente**, deben **referenciar** según el **tipo de cita** correspondiente. Véase Guía en **Sección 4, numerales 4.16 y 4.17**

## 2.1 Resuelvan el siguiente ejercicio usando listas enlazadas:

Estás escribiendo un texto largo con un teclado roto. Bueno, no está tan roto. El único problema es que algunas veces la tecla “Inicio” o la tecla “Fin” se presionan solas (internamente).

Usted no es consciente de este problema, ya que está concentrado en el texto y ni siquiera mira el monitor. Luego de que usted termina de escribir puede ver un texto en la pantalla (si decide mirarlo).

### Entrada

Habrán varios casos de prueba. Cada caso de prueba es una sola línea conteniendo por lo menos uno y como máximo 100 000 caracteres: letras, guiones bajos, y dos caracteres especiales '[' y ']'. '[' significa que la tecla “Inicio” fue presionada internamente, y ']' significa que la tecla “Fin” fue presionada internamente. El input se finaliza con un fin-de-línea (EOF).

### Salida

Por cada caso de prueba imprima la forma en que quedaría el texto teniendo en cuenta que cuando se presiona la tecla 'Inicio' lo manda al principio de la línea, y 'Fin' al final de esta. Asuma que todo el texto está en una sola línea.

### Ejemplo de entrada

```
This_is_a_[Beiju]_text
[] [] [] Happy_Birthday_to_Tsinghua_University
asd[fgh[jkl
asd[fgh[jkl[
[[a[[d[f[[g[g[h[h[dgd[fgsfa[f
asd[gfh[[dfh]hgh]fdhfd[dfg[d]g[d]dg
```

### Ejemplo de salida

```
BeijuThis_is_a__text
Happy_Birthday_to_Tsinghua_University
jklfghasd
jklfghasd
ffgsfadgdhggfda
dddfgdfhgfhasdhghfdhfdgdg
```

## 2.2 [Ejercicio Opcional] Resuelvan el siguiente problema usando pilas:

### Antecedentes

En muchas áreas de la ingeniería de sistemas se usan dominios simples, abstractos para tanto estudios analíticos como estudios empíricos. Por ejemplo, un estudio de IA (inteligencia artificial) en plantación y robótica (STRIPS) usó un mundo de bloques en donde un brazo robótico realizaba tareas que involucraban manipulación de bloques.

En este problema usted va a modelar un mundo de bloques simple bajo ciertas reglas y restricciones. En vez de determinar cómo alcanzar un cierto estado, usted va a “programar” un brazo robótico para que responda a un cierto conjunto de comandos.

### El Problema

El problema es interpretar una serie de comandos que dan instrucciones a un brazo robótico sobre como manipular bloques que están sobre una mesa plana. Inicialmente hay  $n$  bloques en la mesa (enumerados de 0 a  $n-1$ ) con el bloque  $b_i$  adyacente al bloque  $b_{i+1}$  para todo  $0 \leq i < n-1$  tal y como se muestra en el siguiente diagrama:



**Figura 1: Configuración inicial de los bloques de mesa**

Los comandos válidos para el brazo robótico que manipula los bloques son:

1. **move a onto b:** donde  $a$  y  $b$  son números de bloques, pone el bloque  $a$  encima del bloque  $b$  luego de devolver cualquier bloque que estén apilados sobre los bloques  $a$  y  $b$  a sus posiciones iniciales.
2. **move a over b:** donde  $a$  y  $b$  son números de bloques, pone el bloque  $a$  encima de la pila que contiene al bloque  $b$ , luego de retornar cualquier bloque que está apilado sobre el bloque  $a$  a su posición inicial.
3. **pile a onto b:** donde  $a$  y  $b$  son números de bloques, mueve la pila de bloques que consiste en el bloque  $a$  y todos los bloques apilados sobre este, encima de  $b$ . Todos los bloques encima del bloque  $b$  son movidos a su posición inicial antes de que se dé el apilamiento. Los bloques apilados sobre el bloque  $a$  conservan su orden original luego de ser movidos.
4. **pile a over b:** donde  $a$  y  $b$  son números de bloques, pone la pila de bloques que consiste en el bloque  $a$  y todos los bloques que están apilados sobre este, encima de la pila que contiene al bloque  $b$ . Los bloques apilados sobre el bloque  $a$  conservan su orden original luego de ser movidos.
5. **quit:** termina las manipulaciones en el mundo de bloques.

Cualquier comando en donde  $a = b$  o en donde  $a$  y  $b$  están en la misma pila de bloques es un comando ilegal. Todo comando ilegal debe ser ignorado y no debe afectar la configuración de los bloques.

### La entrada

La entrada inicia con un entero  $n$  sólo en una línea representando el número de bloques en el mundo de bloques. Asuma que  $0 < n < 25$ .

Este número es seguido por una secuencia de comandos de bloques, un comando por línea. Su programa debe procesar todos los comandos hasta que encuentre el comando quit.

Asuma que todos los comandos tendrán la forma especificada arriba. No se le darán comandos sintácticamente incorrectos.

### La salida

La salida deberá consistir en el estado final del mundo de bloques. Cada posición original de los bloques enumerada  $0 \leq i < n-1$  (donde  $n$  es el número de bloques) deberá aparecer seguida inmediatamente de dos puntos (:). Si hay por lo menos un bloque en esta posición, los dos puntos deberán estar seguidos de un espacio, seguido de una lista de bloques que aparece apilada en esa posición con el número de cada bloque separado de los demás por un espacio. No ponga espacios delante de las líneas.

Deberá imprimir una línea por cada posición (es decir, habrá  $n$  líneas de salida donde  $n$  es el entero en la primera línea de la salida)

### Ejemplo de entrada

```
10
move 9 onto 1
move 8 over 1
move 7 over 1
move 6 over 1
pile 8 over 6
pile 8 over 5
move 2 over 1
move 4 over 9
quit
```

### Ejemplo de salida

0: 0  
1: 1 9 2 4  
2:  
3: 3  
4:  
5: 5 8 7 6  
6:  
7:  
8:  
9:

2.3. [Ejercicio Opcional] Resuelvan el siguiente problema <http://bit.ly/2j9D1VF>

2.4. [Ejercicio Opcional] Resuelvan el siguiente problema <https://goo.gl/WVXJPx>

### 3) Simulacro de preguntas de sustentación de Proyectos



Véase Guía en **Sección 3,**  
**Numeral 3.4**



Entregar informe de  
laboratorio en **PDF**



Usen la **plantilla** para  
responder laboratorios



**No apliquen Normas**  
**Icontec** para esto



En la vida real, es necesario distinguir cuándo el comportamiento asintótico de un método es constante, lineal o cuadrático, porque la diferencia es enorme para grandes volúmenes de datos como los que se utilizan en análisis de la bolsa de valores, redes sociales y mercadeo

**3.1** Teniendo en cuenta lo anterior, calculen la complejidad de cada ejercicio con cada implementación de listas. Es decir, hagan una tabla, en cada fila coloquen el número del ejercicio, en una columna la complejidad de ese ejercicio usando *ArrayList* y en la otra columna la complejidad de ese ejercicio usando *LinkedList*.


	<i>ArrayList</i>	<i>LinkedList</i>
Ejercicio 1.1		
Ejercicio 1.2		
Ejercicio 1.3		
Ejercicio 1.4		



En la vida real, el trabajo de testing es uno de los mejor remunerados y corresponde a un Ingeniero de Sistemas

**3.2** Teniendo en cuenta lo anterior, verifiquen, utilizando *JUnit*, que todos los *tests* escritos en el numeral 1.2 pasan. Muestren, en su informe de PDF, que los tests se pasan correctamente; por ejemplo, incluyendo una imagen de los resultados de los tests.

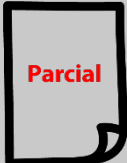
**3.3** Expliquen con sus propias palabras cómo funciona la implementación del ejercicio 2.1 y [opcionalmente] el 2.2

	<p>UNIVERSIDAD EAFIT ESCUELA DE INGENIERÍA DEPARTAMENTO DE INFORMÁTICA Y SISTEMAS</p>	<p>Cód. ST0245 Estructuras de Datos 1</p>
---	---	---


**3.4** Calculen la complejidad del ejercicio realizado en el numeral 2.1 y [opcionalmente] 2.2, y agregarla al informe PDF

**3.5** Expliquen con sus palabras las variables (qué es 'n', qué es 'm', etc.) del cálculo de complejidad del numeral 3.3

#### 4) Simulacro de Parcial en el informe PDF



Para este simulacro, agreguen ***sus respuestas*** en el informe PDF.



*El día del Parcial no tendrán computador, JAVA o acceso a internet.*

**1.** ¿Cuál operación tiene una mayor complejidad asintótica, para el peor de los casos, en una lista simplemente enlazada?

- a) Buscar un dato cualquiera en la lista
- b) Insertar un elemento cualquiera en la lista
- c) Las dos tienen la misma complejidad asintótica

**2.** Pepito quiere conocer el nombre de todos los productos de una tienda. Pepito diseñó un programa que imprime los elementos de la una lista enlazada. La variable  $n$  representa el tamaño de la lista. En el peor de los casos, ¿cuál es la complejidad asintótica para el algoritmo?



**Importante:** Recuerde que el ciclo `for each` implementa iteradores que permiten obtener el siguiente (o el anterior) de una lista enlazada en tiempo constante, es decir, en  $O(1)$ .

```
01 public void listas(LinkedList<String> lista) {  
02     for(String nombre: lista)  
03         print(nombre); }
```

- a)  $O(n^2)$
- b)  $O(1)$
- c)  $O(n)$
- d)  $O(\log n)$

3. En el juego de *hot potato* (conocido en Colombia como *Tingo, Tingo, Tango*), los niños hacen un círculo y pasan al vecino de la derecha, un elemento, tan rápido como puedan. En un cierto punto del juego, se detiene el paso del elemento. El niño que queda con el elemento, sale del círculo. El juego continúa hasta que sólo quede un niño.

Este problema se puede simular usando una lista. El algoritmo tiene dos entradas:

Una lista `q` con los nombres de los niños y una constante entera `num`. El algoritmo retorna el nombre de la última persona que queda en el juego, después de pasar la pelota, en cada ronda, `num` veces.

Como un ejemplo, para el círculo de niños [*Bill, David, Susan, Jane, Kent, Brad*], donde *último* es el primer niño, *Brad* el primer niño, y `num` es igual a 7, la respuesta es *Susan*.

En Java, el método `add` agrega un elemento al comienzo de una lista, y el método `remove` retira un elemento del final de una lista y retorna el elemento.

```
01 String hotPotato(LinkedList<String> q, int num)  
02     while (_____)
```

```
03     for (int i = 1; i _____ num; i++)
04         q.add(_____);
05     q.remove();
06     return _____;
```

A continuación, complete los espacios restantes del código anterior:

**a)** Complete el espacio de la línea 02

\_\_\_\_\_

**b)** Complete el espacio de la línea 03

\_\_\_\_\_

**c)** Complete el espacio de la línea 04

\_\_\_\_\_

**d)** Complete el espacio de la línea 06

\_\_\_\_\_

**4.** El siguiente es un algoritmo invierte una lista usando como estructura auxiliar una pila:

```
01 public static LinkedList <String> invertir
    (LinkedList <String> lista){
02     Stack <String> auxiliar = new Stack <String>();
03     while(..... > 0){
04         auxiliar.push(lista.removeFirst());
05     }
06     while(auxiliar.size() > 0){
07         .....
08     }
09     return lista;
10 }
```

De acuerdo a lo anterior, responda las siguientes preguntas:

**a)** ¿Qué condición colocaría en el ciclo while de la línea 3? (10%)

.....

b) Complete la línea 7 de forma que el algoritmo tenga sentido (10%)

.....

5. En un banco, se desea dar prioridad a las personas de edad avanzada. El ingeniero ha propuesto un algoritmo para hacer que la persona de mayor edad en la fila quede en primer lugar.

Para implementar su algoritmo, el ingeniero utiliza colas. Las colas en Java se representan mediante la interfaz `Queue`. Una implementación de `Queue` es la clase `LinkedList`.

El método `offer` inserta en una cola y el método `poll` retira un elemento de una cola.

```
01 public Queue<Integer> organizar(  
    Queue<Integer> personas){  
02     int mayorEdad = 0;  
03     int edad;  
04     Queue<Integer> auxiliar1 =  
        new LinkedList<Integer>();  
05     Queue<Integer> auxiliar2 =  
        new LinkedList<Integer>();  
06     while(personas.size() > 0){  
07         edad = personas.poll();  
08         if(edad > mayorEdad) mayorEdad = edad;  
09         auxiliar1.offer(edad);  
10         auxiliar2.offer(edad);  
11     }  
12     while(.....){  
13         edad = auxiliar1.poll();  
14         if(edad == mayorEdad) personas.offer(edad);  
15     }  
16     while(.....){  
17         edad = auxiliar2.poll();  
18         if(edad != mayorEdad) .....;  
19     }  
20     return personas;  
21 }
```

a) ¿Que condiciones colocaría en los 2 ciclos `while` de líneas 12 y 16, respectivamente?

.....

b) Complete la línea 18 con el objeto y el llamado a un método, de forma que el algoritmo tenga sentido

.....

6. ¿Cuál es la complejidad asintótica, para el peor de los casos, de la función `procesarCola(q, n)`?

```
public void procesarCola(Queue q, int n)
    for (int i = 0; i < n; i++)
        for(int j = 0; j < n; j++)
            q.add(j);
            //Hacer algo en O(1)
```

- a)  $O(n)$
- b)  $O(|q|)$ , donde  $|q|$  es el número de elementos de  $q$
- c)  $O(n^2)$
- d)  $O(2^n)$

En Java, el método `add` agrega un elemento al comienzo de una cola.

## 5. [Ejercicio Opcional] Lectura recomendada



"Quienes se preparan para el ejercicio de una profesión requieren la adquisición de competencias que necesariamente se sustentan en procesos comunicativos. Así cuando se entrevista a un ingeniero recién egresado para un empleo, una buena parte de sus posibilidades radica en su capacidad de comunicación; pero se ha observado que esta es una de sus principales debilidades..."

Tomado de <http://bit.ly/2gJKzJD>



Véase Guía en **Sección 3, numeral 3.5 y 4.20** de la Guía Metodológica, "Lectura recomendada" y "Ejemplo para realización de actividades de las Lecturas Recomendadas", respectivamente

Posterior a la lectura del texto "**Narasimha Karumanchi, Data Structures and Algorithms made easy in Java, (2<sup>nd</sup> edition), 2011. Chapter 3: Linked Lists.**" realicen las siguientes actividades que les permitirán sumar puntos adicionales:

- a) Escriban un resumen de la lectura que tenga una longitud de 100 a 150 palabras
- b) Hagan un mapa conceptual que destaque los principales elementos teóricos



**NOTA 1:** Si desean una lectura adicional en inglés, considere la siguiente: "**Robert Lafore. Data Structures and Algorithms in Java. Chapter 5: Linked Lists**" que pueden encontrarla en biblioteca



**NOTA 2:** Si desean una lectura sobre Pilas y Colas, atiendan al siguiente texto "**Robert Lafore, Data Structures and Algorithms in Java (2<sup>nd</sup> edition), 2002. Chapter 4: Stacks and Queues.**",



**NOTA 3:** Estas respuestas también deben incluirlas en el informe PDF

## 6. [Ejercicio Opcional] Trabajo en Equipo y Progreso Gradual



El trabajo en equipo es una exigencia actual del mercado. "Mientras algunos medios retratan la programación como un trabajo solitario, la realidad es que requiere de mucha comunicación y trabajo con otros. Si trabajas para una compañía, serás parte de un equipo de desarrollo y esperarán que te comuniques y trabajes bien con otras personas"

Tomado de <http://bit.ly/1B6hUDp>



Véase Guía en **Sección 3, numeral 3.6** y **Sección 4, numerales 4.21, 4.22 y 4.23** de la Guía Metodológica

- a) Entreguen copia de todas las actas de reunión usando el tablero Kanban, con fecha, hora e integrantes que participaron
- b) Entreguen el reporte de *git*, *svn* o *mercurial* con los cambios en el código y quién hizo cada cambio, con fecha, hora e integrantes que participaron
- c) Entreguen el reporte de cambios del informe de laboratorio que se genera *Google docs* o herramientas similares



**NOTA 1:** Estas respuestas también deben incluirlas en el informe PDF

## Resumen de ejercicios a resolver

DOCENTE MAURICIO TORO BERMÚDEZ  
Teléfono: (+57) (4) 261 95 00 Ext. 9473. Oficina: 19 - 627  
Correo: [mtorobe@eafit.edu.co](mailto:mtorobe@eafit.edu.co)

**1.1** Desarrollen un método que reciba una lista y devuelva la multiplicación de todos los números introducidos en una lista. Como un ejemplo, para el arreglo [1,2,3,4], el método debe retornar  $1*2*3*4 = 24$

**1.2** Implementen un método que reciba una lista e inserte un dato de modo similar a la función insertar al final de la lista. La diferencia es que ahora no se debe permitir insertar datos repetidos. Si un dato ya está almacenado, entonces la lista no varía, pero tampoco es un error. Llámelo `SmartInsert(Lista l, int data)`.

**1.3** Escriban un método que reciba una lista y calcule cuál es la posición óptima de la lista para colocar un pivote

**1.4** Elaboren un programa que reciba las neveras, en la forma en que están ordenadas en el almacén, y las solicitudes, según el orden en que llegaron, y que imprima qué neveras del almacén quedan asignadas a cada tienda.

**1.5** Implementen los métodos insertar un elemento en una posición determinada (conocido como *insert* o *add*), borrar un dato en una posición determinada (conocido como *remove*) y verificar si un dato está en la lista (conocido como *contains*) en la clase *LinkedListMauricio*

**1.6** En el código entregado por el profesor está el método *get* y sus pruebas. Teniendo en cuenta esto, realicen 3 *tests* unitarios para cada método. La idea es probar que su implementación de los métodos *insert* y *remove* funcionan correctamente por los menos en los siguientes casos


**1.7** En un banco hay 4 filas de clientes y solamente 2 cajeros. Se necesita simular cómo son atendidos los clientes por los cajeros. Si lo desean, usen su implementación de listas enlazadas; de lo contrario, use la del API de Java

**2.1** Resuelvan el problema planteado usando listas enlazadas

**2.2 [Ejercicio Opcional]** Resuelvan el problema de una serie de comandos que dan instrucciones a un brazo robótico sobre como manipular bloques que están sobre una mesa plana, usando pilas

**2.3. [Ejercicio Opcional]** Resuelvan el siguiente problema <http://bit.ly/2j9D1VF>

**2.4. [Ejercicio Opcional]** Resuelvan el siguiente problema <https://goo.gl/WVXJPx>

	UNIVERSIDAD EAFIT ESCUELA DE INGENIERÍA DEPARTAMENTO DE INFORMÁTICA Y SISTEMAS	Cód. ST0245
		Estructuras de Datos 1

**3.1** Calculen la complejidad de cada ejercicio con cada implementación de listas. Es decir, hagan una tabla, en cada fila coloquen el número del ejercicio, en una columna la complejidad de ese ejercicio usando *ArrayList* y en la otra columna la complejidad de ese ejercicio usando *LinkedList*.

**3.2** Verifiquen, utilizando *JUnit*, que todos los *tests* escritos en el numeral 1.2 pasan. Muestren, en su informe de PDF, que los tests se pasan correctamente; por ejemplo, incluyendo una imagen de los resultados de los *tests*.

**3.3** Expliquen con sus propias palabras cómo funciona la implementación del ejercicio 2.1 y [opcionalmente] el 2.2

**3.4** Calculen la complejidad del ejercicio realizado en el numeral 2.1 y [opcionalmente] 2.2, y agregarla al informe PDF

**3.5** Expliquen con sus palabras las variables (qué es 'n', qué es 'm', etc.) del cálculo de complejidad del numeral 3.3

**4.** Simulacro de Parcial

**5.** Lectura recomendada **[Ejercicio Opcional]**

**6.** Trabajo en Equipo y Progreso Gradual **[Ejercicio Opcional]**

**7.** Entreguen el código y el informe traducido al inglés. Utilicen la plantilla dispuesta en este idioma para el laboratorio. **[Ejercicio Opcional]**



## **Ayudas para resolver los ejercicios**

<b>Ayudas para el Ejercicio 1.....</b>	<b><u>Pág. 26</u></b>
<b>Ayudas para el Ejercicio 1.3.....</b>	<b><u>Pág. 26</u></b>
<b>Ayudas para el Ejercicio 1.4.....</b>	<b><u>Pág. 27</u></b>
<b>Ayudas para el Ejercicio 1.5.....</b>	<b><u>Pág. 28</u></b>
<b>Ayudas para el Ejercicio 1.6.....</b>	<b><u>Pág. 30</u></b>
<b>Ayudas para el Ejercicio 1.7.....</b>	<b><u>Pág. 30</u></b>
<b>Ayudas para el Ejercicio 2.1.....</b>	<b><u>Pág. 31</u></b>
<b>Ayudas para el Ejercicio 2.2.....</b>	<b><u>Pág. 31</u></b>
<b>Ayudas para el Ejercicio 2.3.....</b>	<b><u>Pág. 32</u></b>
<b>Ayudas para el Ejercicio 2.4.....</b>	<b><u>Pág. 32</u></b>
<b>Ayudas para el Ejercicio 3.1.....</b>	<b><u>Pág. 32</u></b>
<b>Ayudas para el Ejercicio 4.....</b>	<b><u>Pág. 33</u></b>
<b>Ayudas para el Ejercicio 5a.....</b>	<b><u>Pág. 34</u></b>
<b>Ayudas para el Ejercicio 5b.....</b>	<b><u>Pág. 34</u></b>
<b>Ayudas para el Ejercicio 6a.....</b>	<b><u>Pág. 34</u></b>
<b>Ayudas para el Ejercicio 6b.....</b>	<b><u>Pág. 34</u></b>
<b>Ayudas para el Ejercicio 6c.....</b>	<b><u>Pág. 35</u></b>

## Ayudas para el Ejercicio 1



**PISTA 1:** Lean las diapositivas tituladas “*Data Structures I: O Notation*”, donde encontrarán algoritmos, y en *Eafit Interactiva*, las implementaciones en Java de cada uno

## Ayudas para el Ejercicio 1.3



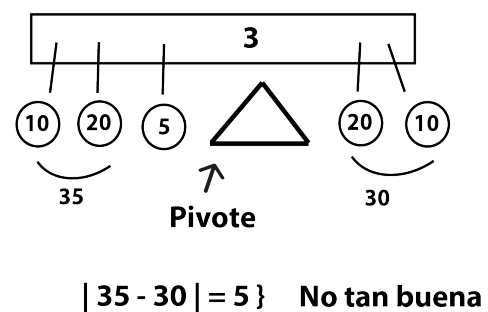
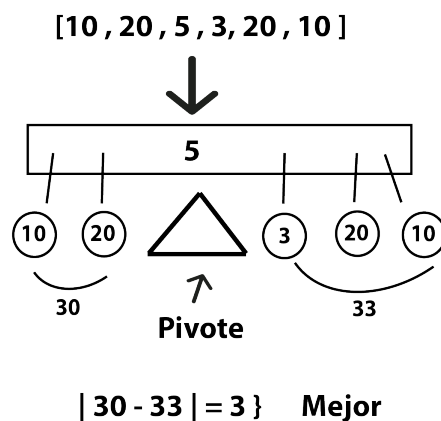
**PISTA 1:** Como un ejemplo, para la lista [10, 20, 5, 3, 20, 10] la posición 2 (en donde está la materia de peso 5 kg) es el mejor pivote, porque al lado izquierdo queda un peso de 33 kg y al lado derecho 33 kg; en cambio, en otras posiciones, la suma de pesos a la izquierda y a la derecha queda más desigual



**PISTA 2:** Como otro ejemplo, para la lista [10, 2, 4, 8] el pivote debería ir en la posición 1, en donde está el peso de 2 kg. Como un final ejemplo, en la lista [10, 2, 5, 2, 11], el pivote debería ir en la posición 2 en donde está el peso de 5 kg



**PISTA 3:**



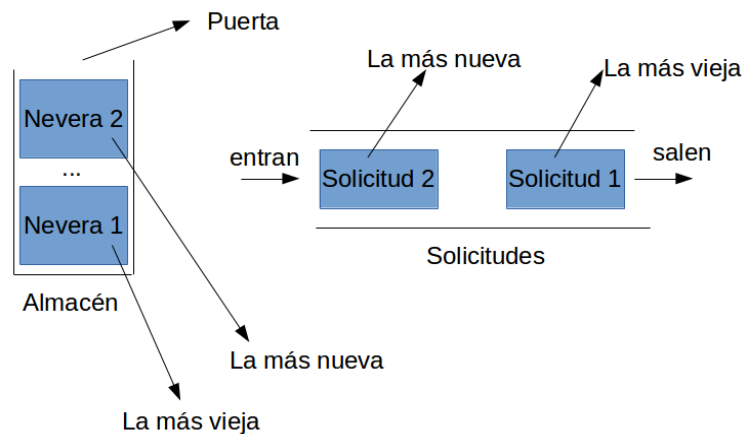
## Ayudas para el Ejercicio 1.4



**PISTA 1:** Utilicen listas enlazadas



**PISTA 2:** No supongan, erróneamente, que hay suficientes neveras para cumplir las solicitudes



**PISTA 3:** Como un ejemplo, para las siguientes neveras y solicitudes,

```
almacen = [(1,"haceb"), (2,"lg"), (3,"ibm"), (4,"haceb"), (5,"lg"),
(6,"ibm"), (7,"haceb"), (8,"lg"), (9,"ibm"), (8,"lg"), (9,"ibm")]
```

```
solicitudes = [("eafit", 10), ("la14", 2), ("olimpica", 4), ("éxito",
1)]
```

Donde “éxito” fue la primera solicitud y “eafit” la última, e “ibm” con código 9 fue la última nevera ingresada a la bodega, la respuesta debe ser:

```
[('exito', [(9, 'ibm')]),
('olimpica', [(8, 'lg'), (9, 'ibm'), (8, 'lg'), (7, 'haceb')]),
('la14', [(6, 'ibm'), (5, 'lg')]),
('eafit', [(4, 'haceb'), (3, 'ibm'), (2, 'lg'), (1, 'haceb')])]
```



**PISTA 4:** Si deciden hacer la documentación, consulten la *Guía en Sección 4, numeral 4.1 “Cómo escribir la documentación HTML de un código usando JavaDoc”*

## Ayudas para el Ejercicio 1.5



**PISTA 1:** Diferencien *LinkedListMauricio* de *LinkedList* del API de Java. Un error común es creer que todo se soluciona llamando los métodos existentes en *LinkedList* y, no es así, la idea es implementar una lista enlazada nosotros mismos. A continuación, un ejemplo del error:

```
// Retorna el tamaño actual de la lista
public int size()
{
    return size();
}

// Retorna el elemento en la posición index
public int get(int index)
{
    return get(index);
}

// Inserta un dato en la posición index
public void insert(int data, int index)
{
    if (index <= size())
    {
        insert(data, index);
    }
}

// Borra el dato en la posición index
public void remove(int index)
{
    remove(index);
}

// Verifica si está un dato en la lista
public boolean contains(int data)
{
    return contains(data);
}
```



**PISTA 2:** Otro error común es pensar que todo se soluciona usando `getNode`. Sí, `getNode` es de gran utilidad, pero ¿qué pasa si `index = 0` o si la lista está vacía? A continuación, un ejemplo de cómo no usar `getNode`.

```
// Borra el dato en la posicion index
public void remove(int index)
{
    getNode(index-1).next=getNode(index+1);
}
```

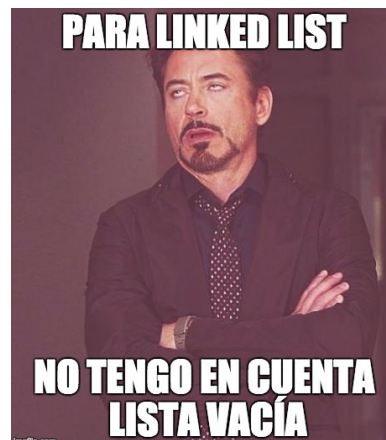


**PISTA 3:** Otro problema común es destruir la lista sin culpa cuando uno desea consultarla. Como un ejemplo, estos métodos de *size* calculan el tamaño pero dañan la lista, dejando la referencia *first* en *null*

```
// Malo porque daña la lista
public int size1() {
    int i = 0;
    while (first != null)
    {
        first = first.next;
        i++;
    }
    return i;
}
```

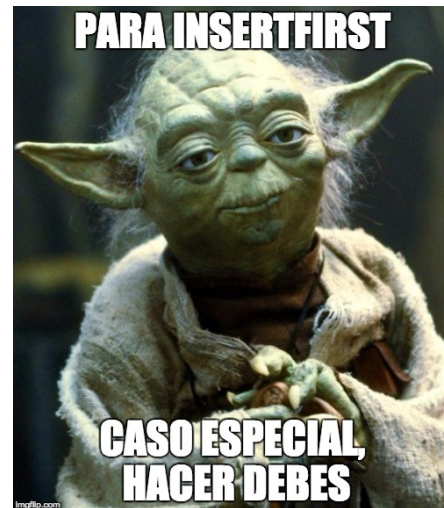
```
// Malo porque daña la lista
public int size2() {
    if (first == null)
        return 0;
    else
        first = first.next;
    return 1 + size();
}
```

## Errores Comunes



DOCENTE MAURICIO TORO BERMÚDEZ  
Teléfono: (+57) (4) 261 95 00 Ext. 9473. Oficina: 19 - 627  
Correo: [mtorobe@eafit.edu.co](mailto:mtorobe@eafit.edu.co)

## Errores Comunes



## Ayudas para el Ejercicio 1.6



**PISTA 1:** Véase Guía en **Sección 4, numeral 4.14** “Cómo hacer pruebas unitarias en BlueJ usando JUnit” y **numeral 4.15** “Cómo compilar pruebas unitarias en Eclipse

## Ayudas para el Ejercicio 1.7



### PISTAS:

1. Realicen primero un dibujo de cada fila del banco y cada cajero
2. Identifiquen si la fila es una lista, cola, pila o arreglo
3. Identifiquen si el cajero es un número, una lista, una pila, una cola o un arreglo
4. Identifiquen si va a guardar las 4 filas en una lista, pila, cola o arreglo.

DOCENTE MAURICIO TORO BERMÚDEZ  
Teléfono: (+57) (4) 261 95 00 Ext. 9473. Oficina: 19 - 627  
Correo: [mtorobe@eafit.edu.co](mailto:mtorobe@eafit.edu.co)

```
major class Laboratory4{  
    public static void simular(??? filas)  {  
        ...  
    }  
}
```



**PISTA 5:** Si deciden hacer la documentación, consulten la *Guía en Sección 4, numeral 4.1* “Cómo escribir la documentación HTML de un código usando JavaDoc”

## Ayudas para el Ejercicio 2.1



**PISTA 1:** Si este ejercicio se hace en un juez en línea con una complejidad de  $O(n^2)$ , saldrá *time out*

Como un ejemplo, de esta forma el recorrido de la lista se convierte en  $O(n^2)$  porque `get()` es  $O(n)$  y está en un ciclo que se hace  $n$  veces.

```
public static int multi(LinkedList<Integer> lista) {  
    int acum = 1;  
    for (int i = 0; i < lista.size(); ++i) {  
        acum *= lista.get(i);  
    }  
    return acum;  
}
```



**PISTA 2:** Véase Guía en *Sección 4, numeral 4.13* “Cómo usar Scanner o BufferedReader”

## Ayudas para el Ejercicio 2.2



**PISTA 1:** Utilicen *pilas*



**PISTA 2:** Véase *Guía en Sección 4, numeral 4.13 “Cómo usar Scanner o BufferedReader”*



**PISTA 3:** Lo mejor es utilizar la técnica de diseño de modularización. Entonces, escribir un método para resolver cada uno de los siguientes problemas:

- ☒ move a onto b
- ☒ move a over b
- ☒ pile a onto b
- ☒ pile a over b

Posteriormente, se crea un mundo de bloques usando una pila para cada columna. Finalmente, se lee la entrada y se ejecuta la acción que corresponda de las 4 definidas anteriormente.

## Ayudas para el Ejercicio 2.3



**PISTA 1:** No construya una solución  $O(n^2)$  para este problema

## Ayudas para el Ejercicio 2.4



**PISTA 1:** Utilicen pilas

## Ayudas para el Ejercicio 3.1



**PISTA 1:** Si quieren que sean eficientes los ejercicios, usen iteradores, así: <http://bit.ly/2cwWdbe>



Como un ejemplo, de esta forma el recorrido de la lista se convierte en  $O(n^2)$  porque `get()` es  $O(n)$  y está en un ciclo que se hace  $n$  veces.

```
public static int multi(LinkedList<Integer> lista) {  
    int acum = 1;  
    for (int i = 0; i < lista.size(); ++i) {  
        acum *= lista.get(i);  
    }  
    return acum;  
}
```



**PISTA 2:** Recuerden que la complejidad de un algoritmo es  $O(n^2)$  cuando hay 2 ciclos anidados que dependen de la misma entrada, pero si hay dos entradas, por ejemplo dos arreglos diferentes, y un ciclo recorre un arreglo y el otro ciclo recorre el otro arreglo, entonces la complejidad es  $O(n.m)$



**PISTA 3:** La complejidad de este algoritmo es  $O(n)$  porque el `for each` es capaz de acceder al siguiente elemento de una lista enlazada en tiempo constante

```
for(String nombre : lista)  
    print(nombre)
```



**PISTA 4:** Véase *Guía en Sección 4, numeral 4.11* “Cómo escribir la complejidad de un ejercicio en línea”



**PISTA 5:** Véase *Guía en Sección 4, numeral 4.19* “Ejemplos para calcular la complejidad de un ejercicio de CodingBat”

## Ayudas para el Ejercicio 4



**PISTA 1 :** Véase *Guía en Sección 4, Numeral 4.18* “Respuestas del Quiz”



**PISTA 2:** Lean las diapositivas tituladas “*Data Structures I: Linked Lists*” y “*Data Structures I: Implementation of Lists*” y encontrarán la mayoría de las respuestas

## Ayudas para el Ejercicio 5a



**PISTA 1:** En el siguiente enlace, unos consejos de cómo hacer un buen resumen <http://bit.ly/2knU3Pv>



**PISTA 2:** [Aquí](#) le explican cómo contar el número de palabras en Microsoft Word

## Ayudas para el Ejercicio 5b



**PISTA 1:** Para que hagan el mapa conceptual se recomiendan herramientas como las que encuentran en <https://cacoo.com/> o <https://www.mindmup.com/#m:new-a-1437527273469>

## Ayudas para el Ejercicio 6a



**PISTA 1:** Véase *Guía en Sección 4, Numeral 4.21 “Ejemplo de cómo hacer actas de trabajo en equipo usando Tablero Kanban”*

## Ayudas para el Ejercicio 6b



**PISTA 1:** Véase *Guía en Sección 4, Numeral 4.23 “Cómo generar el historial de cambios en el código de un repositorio que está en svn”*

## Ayudas para el Ejercicio 6c



**PISTA 1:** Véase Guía en Sección 4, Numeral 4.22 “**Cómo ver el historial de revisión de un archivo en Google Docs**”